Symbolic Two-Dimensional Minimization of Strongly Unspecified Finite State Machines.

M.A. Perkowski +, L. Józwiak †, and W. Zhao +

\+ Department of Electrical Engineering, Portland State University,

P.O. Box 751, Portland OR 97207-0751, U.S.A,

Email: *mperkows@ee.pdx.edu*, Tel. (503) 725-5411, Fax. (503) 725-4882,

† Faculty Electr. Engn., Eindhoven Univ. of Technology,

P.O.Box 513, 5600 MB Eindhoven, The Netherlands, lech@eb.ele.tue.nl

Abstract— A generalization of the classical state minimization problem for Finite State Machine (FSM) is proposed and discussed. In contrast to classical state minimization algorithms that minimize in one dimension (states), our algorithm minimizes the FSM in two dimensions: the numbers of both input symbols and internal state symbols are minimized in an iterative sequence of input minimization and state minimization procedures. This approach leads to an input decomposition of the FSM. A modified formulation of the state minimization problem is also introduced, and an efficient branch-and-bound program, FMINI, is presented. FMINI produces an exact minimum result for each component minimization process and a globally quasi-minimum solution to the entire two-dimensional FSM minimization problem. For some benchmarks, especially those with a high percentage of don't cares, as those that occur in Machine Learning,

this approach produces a more optimum results than those produced by the state minimization alone.

1 Introduction.

Recently, several design methodologies have been proposed, where a control unit is designed as a network of Finite State Machines (presented by De Micheli [5], Devadas and Newton [6, 7], Perkowski and Nguyen [22], and Perkowski and Jóźwiak [25]). Our particular interest is in the methodology from [25], where an *iterative ring network of machines and combinational blocks* is used as a self-controlling data-path. These methods produce *machines with a high percentage of don't cares*, because only for very few combinations of the component machines' states the transitions to their new states are specified. Such machines occur also in practical industrial designs in networks of robotic controllers as for instance the *"control of transportation system, robot manipulator, technological camera and locks for integrated circuits manufacturing line"* or logic control of complex systems such as power stations. **These machines have many don't cares, many inputs and outputs, and many compatible states.**

Another area in which machines with very high percentage of don't cares in both transitions and outputs occur are Machine Learning (ML) and Pattern Recognition [3, 27, 30] in which the percent of don't cares is extremely high and can be above 99%, which makes them very distinct from circuit applications. We proposed a methodology based on state machine acquisition, minimization and decomposition/assignment in [26] for use in Data Mining and Learning Robotics applications. Two-dimensional state minimization is one of its important steps.

Tables with many don't cares are also created as initial specifications for asynchronous machines [13], for machines designed from waveforms, ma-

chines created from partitioning (Devadas [6]), from counter-embedding, and from "state-selected-input" decompositions. Also, sequential don't cares are extensively used. Other methods to create machines with many don't cares have also been reported by Ashar [1, 2] and Devadas and Newton [7].

Several problems related to synthesis, verification and testability of such networks of machines require checking the compatibility of states and detecting the unreachable states. One of the processes traditionally used to verify and optimize hardware-realized machines with compatible states and don't cares is *state minimization of FSM.* It was shown, for instance, by Pager [17] that the problem of determining the minimal closed table for a compiler which uses Knuth's LR(k) parsing algorithm can be reduced to that of finding the minimal closed cover for an incompletely specified flow table. Such tables have usually a high percentage of don't cares.

In this paper, we will be dealing with the new process of state/input minimization, called also two-dimensional (2D) minimization, which gives good results especially when applied to machines with a high percentage of don't cares. The results of this research can find applications in all areas where one has to synthesize, test and verify state machines with don't cares; for instance in industrial controllers, DSP controllers, systolic arrays, cellular automata, protocol engines, or Machine Learning applications.

2 Previous and Current Research on State Minimization.

The idea of **minimizing states and columns of an FSM concurrently** was first introduced by Luccio [15] and by Grasselli and Luccio [10]. The paper by Luccio [15] formulates the problem and gives examples but does not give the solution method. The paper by Grasselli and Luccio [10] gives a complete exact, nearly exhaustive algorithm which has, however, not been programmed. It is our feeling that, if programmed, their algorithm would be very slow for the practical-sized machines that we are concerned with here. Their approach does not consider also the preprocessor circuit design problem, presented below, which is a byproduct of our approach to two-dimensional minimization. Their apprach was never much used or referenced.

Finite State Machine Minimization Theory was developed in the 1960s and 1970s [18, 9, 10, 15, 16, 17]. Unfortunately, until recently, these research efforts had very little, if any, impact on the design practices in the industrial environment and on the CAD systems. State minimization of FSMs has become an active Logic Synthesis research topic again since 1989, because new design methodologies such as partitioning [6], collapsing, parallel graphs [21], and concurrent machines [22] produce FSMs with equivalent states and/or don't cares. Such machines are then minimized, or restructured using other methods, such as Mealy $\leftrightarrow$ Moore transformations, and the "state-selected-input decomposition" (in which the machine is transformed to a form where internal states multiplex subsets of inputs for the transitions executed in those states). More industrial companies are also

now interested in new tools for Programmable Logic Devices (PLDs) and other system/logic design tools that use schematic capture of waveforms, Petri Nets, and concurrent state machines as initial high-level system descriptions. Machines created from such descriptions are highly non-minimal, so programs to minimize FSMs are used, or are planned to be used, in those tools. State minimization is also useful for verification and testability purposes for several types of machines. For the reasons mentioned above, the state minimization problem is recently of an increased interest [1, 2, 27, 28]. State minimizers have been included to the U.C. Berkeley systems such as SIS, and are also important components of verifiers.

The original contribution of this paper is the formulation and realization of a topic that has not been extensively studied until now: **two-dimensional minimization.** No papers other than those by Grasselli and Luccio [10, 15] exist, and there is no information whether their idea has been ever programmed.

Our program, FMINI, has several options, including two-dimensional minimization, next-state selection, concurrent minimization and assignment, equivalent states, Mealy $\leftrightarrow$ Moore conversion, proper graph coloring and closed graph coloring for very large machines, encoder design, and state splitting. The presented approach generalizes some ideas of symbolic minimization of logic functions and state machines proposed in papers of Perkowski et al [21, 22, 19, 26, 14], Ciesielski et al [12, 31], and Devadas et al [2, 1, 6, 7, 8]. In this paper only the two-dimensional minimization and some aspects of the encoder design issues are presented.

3 A Two-Dimensional Minimization Program.

3.1 Overview of Program FMINI.

As a continuation and improvement of the listed above efforts, we developed a synthesis methodology [21] in which an additional abstract minimization process precedes the logic synthesis phase. This new process combines the minimization of the internal states with that of the input states. Each state minimization or input minimization process is performed independently, but such processes are executed in an iterative sequence until no better machine is found. Such an approach leads to a partitioned realization of the initial FSM with an input encoder and the main FSM (Fig. 1).

Our program for this method, called **FMINI**, produces a minimum result for each component minimization process, and a globally quasi-minimum solution to the two-dimensional FSM minimization problem. By two-dimensional we mean iteratively optimizing states and inputs, until no further optimization is possible. The input minimization, from which the whole iterative process starts, is described in section 3.4, and the state minimization is described in section 3.5. Since the entire process produces a partitioned realization of logic, the input encoder design is presented in section 4. We will use a single complete example to illustrate all these steps.

Almost all FSM state minimization algorithms begin with *a state table.* An example of such a table is shown in Table 1. Observe, that columns correspond to disjoint cubes on input bits a, b, c shown below the column labels X_i. Obviously, two next states from some columns are *combinable* when their corresponding next states and outputs are *consistent.* Such two

state entries can be merged. If all pairs of states from two columns can be merged, these entire columns can be combined to a single column. For instance, columns X_2 and X_5 are compatible and can be merged to one column. In input (column) minimization, we attempt to combine the columns to as few groups as possible, where the groups do not overlap. The merging occurs more frequently when there are many don't care terms in the internal state transitions and outputs. The *input column minimization* should then be attempted during the entire state minimization process of the FSM. The process of creating a partition to the minimum number of non-overlapping cliques of columns will be called the *input minimization process.* Let us observe that the input minimization tries to combine the *columns* of the FSM table, while the *state minimization* tries to combine the rows of this table in a standard way. The possibilities of input minimization should also be considered in the new state tables created by each of the state minimization processes mentioned above. Following each input or state minimization procedure, the program tests whether further minimization is possible. The optimal solution results actually include two components: the optimal input column partitioning (OMCCI) and the optimal closed and complete covering (OMCCP) which are the last input column partitioning (MCCI) and the minimal closed and complete state covering (MCCP) created when further minimization is impossible.

In contrast to Grasselli and Luccio [10], we simplify the input minimization problem in **FMINI** by temporarily separating it from the state minimization and reducing it to the *clique partitioning problem.* Although NP-hard as well, this problem is known and several very efficient algorithms

have been proposed for it. Therefore, the two-dimensional minimization in **FMINI** is reduced to a sequence of two simpler NP-hard problems: *clique partitioning* and *binate covering* (called also *covering/closure*). The second problem is solved by the *generation of all cliques (all compatibles),* and the *covering/closure problem of internal states with compatibles* [13].

Concluding on the difference of our approach versus Grasselli and Luccio. Instead of solving a single problem of concurrent combining of any pairs of columns and pairs of rows, which problem, although leads in theory to the truly **exact** minimum size of the table, is extremely complicated, we iterate solving simpler problems for which efficient algorithms are known. Moreover, it is not sure that the minimum size table as proposed by Grasselli and Luccio is a good quality metric, and we assume a more complex and technology-related cost function to be approximately minimized.

The generation of compatibles in **FMINI** is done by a modified fast algorithm of Stoffers [29], but in addition we generate at the same time all compatibles implied by these groups [19]. We concentrated on fast solutions to two problems: (1) clique partitioning, and (2) our modification of the covering/closure problem, because these two problems are the most important to the overall success. This approach allows us also to utilize the specifics of the problem (high percent of don't cares) to the greatest extent.

The *covering/closure problem* (called also binate covering in [28]) is a generalization of the well-known *covering problem* and finds applications also in (three-level) TANT circuits, multi-level logic design, and many other related problems, so very efficient algorithms have been created for it. It was solved by special methods, CC-table methods, integer programming meth-

ods, and implicit methods [9, 10, 15, 16, 28]. **FMINI** employs a *branch-and-bound* tree searching approach with a *cost function CF* to cut off those tree branches which are unable to lead to a Minimal Closed and Complete State Covering (MCCP). The algorithm itself uses the theory of heuristic search in OR-trees [20], based on *heuristic functions* and *perceptron-like automatic learning* of the best quality function to evaluate partial solutions [23]. Its basic idea is to use heuristic knowledge derived from the state table and based on the definitions of groups of states to guide the search in the space of all MCCPs, use chains of implied compatible groups (defined in section 3.3), and use quite sophisticated methods to cut-off the branches as early as possible. The algorithm operates directly on groups and their implications and not on logic or integer equations, which allows to use special heuristics to a larger extent. Confusion can arise from the fact that on one hand we use *heuristics,* and on the other hand we claim *exact minimal solutions.* There is no contradiction here, since heuristics are used only to guide the search, and make it more efficient. Cutting off branches is still done using the cost function which is not heuristic [20]. In the worst case, the entire space is searched, but this happens very rarely, while several **exact** FSM minimizers search the entire space or a large subset of it. Although several papers (including the early ones by Grasselli and Luccio [9, 10]) introduce rules that may be used for backtracking, it seems that these rules were actually not implemented in the published programs. The paper by House and Stevens [11] includes cutting-off rules, but the method of CC-tables combined with integer programming is applied, which makes their method applicable only to small machines (an example with 22 states, 4 inputs

and 2 outputs is given). Rho [28] and the mentioned earlier U.C. Berkeley researchers solve CC-tables with smart data structures, which allows them to solve very large machines, including implicit methods. Although their algorithms use several cut-off rules, still more powerful searching and backtracking heuristics could be perhaps implemented with their superior implicit problem representation, but it would be difficult. Some of the existing exact methods cannot be used for machines with very large sets of compatibles, while the approach used by us searches implicitly and thus never generates all compatibles at one time. Because of the limited space, the search algorithm is not presented in this paper. Here we will concentrate on the entire minimization/decomposition problem formulation, necessary definitions, examples, results, and interpretation.

FMINI uses a weighted cost function to decide if the search on a certain branch (set of compatible groups) should be retained or not [19]. The *weighted cost function CF(S)* of a solution set S of *compatible groups (CG)* in the FSM state minimization is:

$$CF(S) \;=\; a_1 \;*\; CARD(A)\; CARD(S) \;+\; a_2 \;*\; \sum_{S_j \;\in\; S} CARD(S_j) \quad (1)$$

In formula (1), A represents a set of all the internal states of a certain FSM. CARD is the number of elements in the set, a_1 and a_2 are coefficients. The first component of (1) corresponds to the *minimization of the number of states* and the second component to the *minimization of the transition functions,* which is in turn expressed by the minimization of the number of states that occur in more than one CG. This leads to the *maximization of the don't care terms* in Boolean functions when the machine is minimized,

its internal states are assigned with codes, and the excitation functions are finally found. The idea is to preserve all existing don't cares, and even add don't cares in the process, if possible. If coefficients a_1, a_2 satisfy $a_1 >> a_2$, then the algorithm selects the solution which has the minimum second component among all machines with the minimum number of states. When set SMAX can be calculated, coefficients a_1 and a_2 from formula (1) are described by the following formulas:

$$a_1 = \frac{1}{CARD(A)} \quad , \tag{2}$$

$$a_2 = \frac{1}{\sum_{S_j \in SMAX} CARD(S_j)}$$

In the above formula, set *SMAX* denotes the *set of all compatible groups.*

This formulation of cost function for "don't care related" minimization is lacking in other programs and leads to an improved quality of realizations in FMINI. It has special advantages in the 2D minimization because of preserving don't cares. It can be easily verified on some examples given by Grasselli and Luccio (1966) [10, 15] that while a machine state-minimized according to Grasselli/Luccio's method cannot be input-minimized (the case of row reduction preventing column reduction), the same machine state-minimized according to our approach can be still input-minimized leading to the minimum global solution.

3.2 Input/Output Formats and Notation.

FMINI accepts two different input data formats, **Kiss** (state transitions - De Micheli [4]) and **Stab** (state tables with disjoint columns - Kohavi [13],

Perkowski and Liu [21]). The conversion of the **Kiss** and **Stab** formats is a part of the **FMINI** program. The **Kiss** or **Stab** data file represents a completely or incompletely specified FSM. The processes of both input and state minimizations are based only on the **Stab** formatted Mealy state table. A **Kiss** formatted file is converted into **Stab** format before the two-dimensional minimization. The optimal result can be generated in **Kiss** or **Stab** format for the convenience of the next design stages.

FMINI handles multi-bit input and output machines so that such states can be represented by both symbols and binary strings. This is done not only to improve the performance of testing the output consistency in both input and state minimizations but also to improve the performance of symbolic FSM assignment. Each cell of the state table will be referred to by: NS/output. For example, 3/-1 is at the fifth column and fourth row of Table 1.

The *don't-care* terms mean that these bits of the binary representation may have values '1' or '0'. In **FMINI**, the character '-' represents the don't-care term for outputs and the not used variable for input cubes. In the next states, '0' will represent the "don't-care states" and the positive integers will represent the states that have certain specified values.

3.3 Definitions.

Definition 1. A group of present states(PS) $(S_i, ..., S_j)$ of machine M is called a *state compatible group,* if and only if, under every input column X_r $(1 \leq r \leq mx)$ the next states $S'_{i,r}$, ..., $S'_{j,r}$, corresponding to all of the present state rows in this group $(S_i, ..., S_j)$, are either *compatible* (the

corresponding next states(NS) of the group belong to their parent present states or are don't cares) or *consistent* (the corresponding next states of the group have the same value or are don't cares) and the corresponding outputs $Z'_{i,r}$, ..., $Z'_{j,r}$ are *consistent bit by bit* (every corresponding bit of the corresponding elements of the groups have the same value or is a don't care bit).

Definition 2. A group of input states (columns) $(X_i, ..., X_j)$ of machine M is called an *input combinable group (CGI),* if and only if, under every present state row S_r $(1 \leq r \leq na)$ the next states $S'_{r,i}$, ..., $S'_{r,j}$ corresponding to all the input columns from this group $(X_i, ..., X_j)$, are consistent, and the corresponding outputs $Z'_{r,i}$ and $Z'_{r,j}$ are also *consistent bit by bit.*

Definition 3. For a group of present states $(S_i, ..., S_j)$, if under a certain input column X_r the next states $S'_{i,r}, ..., S'_{j,r}$ corresponding to all the present state rows in this group $(S_i, ..., S_j)$ belong to another present states group $(S_p, ..., S_q)$, then the group $(S_i, ..., S_j)$ is considered to *imply* the group $(S'_{i,r}, ..., S'_{j,r})$. If $(S_p, ..., S_q)$ is compatible, then $(S'_{i,r}, ..., S'_{j,r})$ is called an *implied compatible group* (ICG).

Definition 4. A set of compatible groups is said to *overlap* if the same elements appear in different groups of this set.

The idea of our state minimization method is to take smaller compatibles in order to minimize the overlap of states. This leaves more don't cares in the cells of the state table for next state/input minimizations. Moreover, this retains more don't cares after minimizations, which allows for better results during the state assignment [22, 14], decomposition [24], and logic

minimization processes of the entire FSM design.

Definition 5. A set of compatible groups of present states of machine M satisfies the *closure condition,* if each of its *implied compatible groups* is also included in a group from this set as well.

Definition 6. A set of compatible groups of machine M is said to satisfy the *completeness condition,* if each internal state of M is contained in at least one group of this set.

The quality of a compatible group, CG, is defined by Q, the number of its elements.

Definition 7. A set covering which satisfies the conditions of closure and completeness is called a *closed and complete covering.* It is called a *feasible solution.*

The feasible solution that has the lowest *cost,* CF, is called the *exact solution.* The final state table with the exact minimum number of states, but being equivalent to the initial table before this state minimization, can be easily found from the exact solution [13].

From the FSM table and sets of compatibles one can create a new table by randomly selecting one state from every compatible group, and next assigning codes to states. However, better results are obtained if the next-state selection (mapping) process is taken into account, before or together with the state assignment [14, 28]. For the *state minimization,* the compatibility of all the rows in the state table must be tested as follows.

3.4 The Input Minimization Process of FMINI.

An example of a state table for a Mealy machine M^0 is shown in Table 1. It will be used to illustrate the entire minimization process. It has six columns, X_1 ,..., X_6, which means that the machine has six distinguishable input states. There are three primary input signals denoted by *a, b* and *c.* The number of disjoint cubes for n input bits is usually much smaller than 2^n. Input state X_2 corresponds to input cube 10-, meaning a product $a\overline{b}$ of input signals. The table has four rows, internal present states S_1 ,..., S_4. For performing the input combinational logic encoding, inputs are marked by (disjoint) binary cubes (the second row of the column headers). In the input minimization they can be also represented by decimal numbers shown as the subscripts of inputs as well as the addresses I_1 ,..., I_6. Each time that the input or state minimization is completed, a new numbering system of the state table is created by the algorithm to replace the old one.

At first, **FMINI** tests the column compatibility conditions (Definition 2). The condition for NSs is that two inputs are compatible, if in the two columns corresponding to them both of the two corresponding NSs are either the same or one or both of them are don't-care states. Note, that this is different from Grasselli and Luccio [10, 15], where conditional compatibility is considered for columns and rows concurrently.

For the outputs $Z'_{i,j}$ the consistency condition is the same for both processes, i.e. two input columns (or PS rows) are compatible under the condition that every two corresponding outputs must be either the same or one or both of the two bits of outputs are '-'s. This rule must be applied to every two corresponding bits of these two tested outputs. For instance, as

shown in Table 1, the inputs X_2 and X_5 are compatible under the condition that the output bits $Z'_{2,j}$ and $Z'_{5,j}$ $(1 \leq j \leq na)$ must be either the same or at least one of them is '-'. In the input columns two and five $Z'_{2,4}$ = '–' and $Z'_{5,4}$ = '- 1'. Obviously, the first bits of these two terms are '-' and '-'. Since the symbol '-' can represent either '1' or '0', these two '-'s can have the same value. The second bits of these two terms are '-' and '1'. Following the description for the first bit, the '-' here should have the value '1' instead of '0'. Therefore it has the same value '1' with the second bit of $Z'_{5,4}$. As a result, $Z'_{2,4}$ and $Z'_{5,4}$ are consistent. This test is done from j = 1 to $j = na$. As the result of the first input minimization, partitioning to maximum compatibles $\{X_1\}$, $\{X_4\}$, $\{X_2, X_5\}$ and $\{X_3, X_6\}$ of columns was found, thus leading to a new machine M^1 in Table 4.

FMINI maintains a list CGG to accumulate the numbers of compatible input columns. Every time the feasible solution to input minimization is generated, CGG will rearrange these column numbers into the new groups. An example of this accumulation is illustrated in Table 2. New numbers are consecutively assigned to the inputs indicated by binary numbers shown in the headings of Table 1, Table 4, and Table 6. The original Table 2(a) of CGG lists every binary input in each category corresponding to the input addresses I^0 of machine M^0. Every time, when an input minimization process is finished, the input addresses will be distributed in a new category group in CGG, according to the solution. For instance, it has been found that the binary inputs I_3^0 and I_6^0 are compatible after the first input minimization process. The binary inputs I_3^0 and I_6^0 are therefore rearranged in a new address I_4^1, shown in Table 2(b). The number of the inputs in this

group, CARD$\{X_3,\ X_6\} = 2$ is placed in array CFC^1. For the same reason, I_2^0 and I_5^0 are arranged in address I_3^1. After the second execution of the input minimization (Table 6) another new input system is generated and indicated by the address I^2 in Table 2(c). The new binary input addresses I_1^1 and I_4^1 belong to a compatible group. The contents of CGG_1^1 and CGG_4^1 are accumulated into another new group in address I_3^2 of Table 2(c). When the entire iterative machine minimization process is finished, these classes of grouped inputs are part of the final solution and will be dealt with in an input signal encoding procedure for the decomposed logic.

3.5 The State Minimization Process of FMINI.

FMINI uses a merger list COMPAT(i, j) instead of the triangular merger table of the classical method [13]. It is used to confirm the result of searching the compatibility of every pair of inputs in input minimization, or PSs in state minimization. The merger list COMPAT is the input to the clique partitioning, both for columns and for rows. For instance, the first resultant optimal partitioning (OMCCI) for columns is: $\{X_1;\ X_4;\ X_2, X_5;\ X_3, X_6\}$. Next list COMPAT is cleared and reused for the row minimization problem. The particular list COMPAT(i, j) in Table 3 has been derived from the **Stab** formatted state table in Table 4.

The decimal numbers in the first column of the merger list COMPAT indicate the checked input pairs. The incompatible pairs are, both for columns and for rows, indicated by '-1', and all compatible pairs are '1', shown in the second column of this list. All of the numbers in the third (double) column, and next three double columns, represent compatibility conditions.

Suppose a pair of inputs are X_i and X_k. The test of compatibility of the NSs starts from $S'_{i,1}$ and $S'_{k,1}$, and then $S'_{i,2}$ and $S'_{k,2}$ and so on. In input minimization, if there exists any two positive numbers in a certain row of COMPAT, this pair of inputs cannot be compatible. As a result, the second column of these rows will simply be marked by '-1's.

After the *input minimization,* the new state table M^1 has been created (Table 4). Now the task is to execute the *state minimization.* For the *state minimization,* the compatibility of all the rows in the state table must be tested as follows (Table 3). For the compatible group of PS rows, every pair of the NSs and the outputs at the corresponding intersections of the rows must satisfy the compatibility condition specified in Definition 1, or they must satisfy the implied compatible condition specified in Definition 3. The first step for state minimization is to test the compatibility conditions. The array from Table 3 contains all the compatible PS pairs, derived directly from the result of the input minimization, the state table Table 4. For instance, the first row Table 3 informs that states 1,2 are compatible under condition that states 1,3 and states 1,4 are compatible. Observe that Table 3 can be obtained directly from Table 4 in a fast process. If a pair of PS satisfies Definition 1, that is, if the outputs from each pair $(Z'_{i,j},\ Z'_{k,j})$ of PS pair $(S_i,\ S_k)$ are consistent $(Z'_{i,j}\ =\ Z'_{k,j})$, and each pair of next states successors $(S'_{i,j},\ S'_{k,j})$ are either equal, $S'_{i,j}\ =\ S'_{k,j}$ (some of S' or Z' could be don't-care term), or belong to their parent states S_i or S_k, these two rows $(S_i,\ S_k)$, are *strongly compatible (non-conditionally compatible).* Such a group of PSs is called a *strongly compatible group (SCG).* Otherwise, according to Definition 3, if some pairs of the NSs belong to some other

strongly compatible present state pairs, i.e. $(S'_{i,j},\ S'_{k,j}) \in (S_p,\ S_q)$, and others satisfy the requirement of strong compatibility, these two rows are called *implied compatible* (Definition 3). Such a group of PSs is called an **implied compatible group (ICG).** Another case is when some pair of next states $(S'_{i,j},\ S'_{k,j})$ is included in an incompatible state pair $(S_p,\ S_q)$, the set of PSs, $(S_p,\ S_q)$ is **weakly incompatible.** The last case is when outputs are inconsistent, the PS pair (S_p, S_q) is *strongly incompatible.* The last case is when some corresponding bits of the outputs $(Z'_{i,j},\ Z'_{k,j})$ of two PS rows $(S_i,\ S_j)$ are inconsistent (some corresponding bits of the two outputs are '1' and '0') the PS pair (S_p, S_q) is *strongly incompatible.* All such groups are treated in respective special ways by the search algorithm.

In the second column of Table 3, the strongly compatible pairs are marked by '1'. The implied compatibles are marked by '0'. Both the weak and strong incompatibles are marked by '-1'. The implied compatible can be chosen as the CGs of a solution in state minimization if the solution that includes the ICGs satisfies the closure condition. The '0's in second column of Table 3 are the marks for the closure test. For contrast, ICGs cannot be a part of the solution in input minimization; therefore, such pairs of inputs are marked by '-1', as shown in the second column of Table 3.

The following steps are executed for state minimization:

(1) All compatible groups are generated: {1, 2, 3, 4, 12, 13, 14, 23, 34, 123, 134}.

(2) All the minimal closed coverings MCCP for PS rows are created.

The exact minimal closed covering for PS rows (MCCP) is found. The

above procedure, applied to Table 4, creates two solutions $MCCP_1 = \{134, 2\}$ and $MCCP_2 = \{14, 23\}$, after the second phase.

In this particular example, all of the CGs of both the MCC_1 and MCC_2 have the same quality, and the solutions MCC_1 and MCC_2 both have the same cost (the cost value of a solution depends on the number of compatible groups it includes. Obviously, the fewer their number, the better is the solution). During the tree searching phase, $MCCP_2$ is created later than $MCCP_1$. To shorten the execution time of searching for the solutions, we use here only the last exact solution found. Therefore, the last created exact solution $MCC_2 = \{14, 23\}$ is taken, and the new state machine M^2 with new state numbers is generated (Table 5).

The way of arranging a new state table is as follows. Since the new table includes two states covering those four states from the old table, the internal present states of the new table are marked 1' = {1, 4} and 2' = {2, 3}. Therefore, each old state number of NSs must belong to either 1' or 2'. Simply, NS(1') = NS(1, 4) and NS(2') = NS(2, 3) are illustrated in the new state table M^2 (Table 5). Consequently, the outputs of the new table are formed according to the rules: a symbol combined with a don't care gives a symbol. A don't care combined with a don't care gives a don't care.

This result may not be the final one, because the new symbolic system may create some columns which will satisfy the column compatibility condition. If so, this table is returned to the input minimization procedure again (as it was when Table 6 was created), and then to the state minimization, and so on. This iteration will be halted only when there is absolutely no more minimization possibility and the most simplified form of FSM, M*, is given

out. In the example discussed here, the Table 6 is such an optimal machine M*, which has three input columns and two PS rows and cannot be further state-minimized. This state table is equivalent to the initial state table M^0 shown in Table 1.

4 Input Logic Encoding Process of FMINI.

As already discussed, the product implicants (represented as cubes) of primary input variables are accumulated in CGG lists corresponding to the columns of the state table. After two input minimization processes, the CGG for our example is $CGG^2 = \{4, 25, 136\}$, as shown in Table 2(c). For the combinational logic synthesis, the binary input symbolic expressions are needed. Assuming order of variables *a, b, c* in cubes and the grouping of initial column numbers as in CGG^2, the input symbols X_i correspond to lists of prime implicant cubes. For instance, in Table 6, input column X_4 corresponds to cube 001, input column $X_{2,5}$ to cubes 10- and 110, and input column $X_{1,3,6}$ to cubes 000, -11, and 010. Since the primary inputs X_i are divided into three distinguishable groups in the solution set, the outputs of the encoder (which means, the secondary inputs of the FSM) should have at least 2 bits, those signals are denoted here by n and m. This encoding could use any kind of code. Some minimizing possibilities result also from the input-output encoding process to select this code, but code selection is beyond the topic of this paper. The code generated for our case is: $X_4 = (m = 0, n = 0)$, $X_{2,5} = (m = 0, n = 1)$, $X_{1,3,6} = (m = 1, n = 0)$, as in Table 7.

Assuming this code, the logic specification of the encoder block is created

from lists of cubes X_4, $X_{2,5}$ and $X_{1,3,6}$, where $F_m(a,b,c)$ and $F_n(a,b,c)$ correspond to functions on the outputs m, n of the encoder (see Figure 2). Thus, all cubes in list $X_{1,3,6}$, for instance, obtain output value cube 10. The Boolean minimizer Espresso-mv is called to quickly find a quasi-minimal solution for the input encoder. After logic minimization the logic equations of the encoder are as follows: $m = \overline{a}\,\overline{c} + b\,c$, $n = a\,\overline{c} + a\,\overline{b}$. This way, in the entire process thus completed, both the numbers of input bits and state bits of the main FSM have been reduced. and the decomposed realization as in Figure 1 has been generated.

The same code from Table 7 is next applied to the main FSM, and the choice of this code influences the state assignment of the main FSM. Other logic minimizing and don't care utilizing possibilities also result from the quasi-optimal encoding of the encoder outputs and the state/input assignment of the minimized state table [22, 4]. Both the state assignment of the main FSM and the output encoding of the encoder block are related to the way in which the initial machine was minimized. For instance, the *concurrent state minimization and state assignment* process, [14, 28], that leads to substantial improvements can be used. Discussion of these issues is beyond the topic of this paper, let us only remark that all these processes are strongly interrelated.

5 Evaluation of FMINI.

We performed a series of experiments with **FMINI** program using more than 40 FSM benchmarks with 50 or less internal states and 20 or less input bits. The results of these experiments show that **FMINI** can efficiently

handle large scale machines. Some successful results of using **FMINI** for minimization of machines with many don't cares are presented in Table 8. Two-dimensional minimization gives essential size improvements for machines with many (more than 30%) don't cares in their outputs.

In another set of tests, machines from high-level behavioral HDL compiler were minimized. These are machines with a small percentage of don't cares. In three out of 12 examples the two-dimensional minimization was useful. It was also shown to reduce the total layout area. We observed that for machines with a high percentage of don't cares there was more gain from the two-dimensional minimization. Although 2D minimization gives improvements *only on some machines,* it should be always applied because sometimes it may bring improvements that are not achievable without this process. For instance, optimal assignment on a non-minimized machine gives worse results than the same assignment on a minimized one. The improvement on these machines was significant, but on some other benchmark sets, not shown here, it was only marginal or none. In a few cases with identical or nearly identical columns the 2D minimization was useful for machines with a low percentage of don't cares as well.

FMINI generates the optimal solutions more quickly if the percentage of don't care terms in a machine is lower, for instance, below 10%. But then, the cost improvement is not that significant. When the percentage of don't care terms increases, the time needed to generate the OMCCPs will grow, but the best results may be found. In fact, as **STAMINA** [28], **FMINI** gives exact optimum solutions (states only) for machines with the above-mentioned sizes and with a relatively high percentage of don't care

terms.

The user of FMINI can select either the pure state minimization or two-dimensional minimization. Therefore, the concept of 2D minimization, implemented here for the first time, is a useful design alternative for machines with many don't cares and compatible states. 2D minimization should be included in comprehensive design automation systems as one of several, script-selected, design methods. Since it is fast, it can be tried on every designed machine without sacrificing much design time. In most cases, 2D minimization of the number of state and input symbols of an FSM is beneficial even if the numbers of flip-flops and/or input bits of the main FSM are not reduced, because the reduced main machine has more don't cares, which in turn leads to better encodings, state assignments, decompositions, and logic synthesis. The don't cares allow also all the subsequent design stages (such as the state assignment, test generation or logic synthesis) to be performed more efficiently, because the respective procedures work more efficiently on machines with smaller numbers of state and input symbols.

For some larger benchmarks **Kiss** format cannot be converted to **Stab**, so 2D minimization cannot be executed. It can be however observed that some large machines from real life have a small number of input variables or input bits, which makes the 2D minimization method still applicable to them. Moreover, the method is applicable to tables where the columns correspond not to combinations of input signals and where the number of columns grows exponentially with the number of such signals, but to "symbolic inputs" or "input symbols". Such inputs are naturally disjoint and their number is not excessive for tables of real-life machines. Tables of

this type exist for pulse-mode asynchronous machines, microprogrammed machines, and the above mentioned LR(k) parsers. In such cases the input symbols can be still combined in 2D minimization to groups, but the pre-processing logic encoder is not created by FMINI. For all the above types of machines, such grouping has a direct interpretation of "OR-ing", which simplifies the corresponding circuits, or reduces the parser data structures.

6 Conclusion.

FMINI is an efficient state minimizer, especially for machines with a high percentage of don't cares. The concept of two-dimensional minimization, introduced here and implemented for the first time in a computer program, is a useful design alternative for machines with many don't cares and many compatible states. It allows to minimize machines in two dimensions, thus decomposing and totally restructuring the machine. For some machines this can lead to further area minimization with respect to the one-dimensional minimization.

Although each component minimization process in the sequence of input and state minimization procedures is exact, there exist examples [10] which prove that our final machine is not an exact optimum, since minimization in one dimension can prevent the minimization in the other one.

It can be observed, that for many machines taken from industry, there are no possibilities of column or row reductions. However, although our method not always gives an improvement, it can be tried on every designed machine without sacrificing much design time, because it is quite fast, so that there is no risk of incorporating it as one of the several design methods

in a comprehensive design automation system (in the worst case, the same machine will be returned as received). We believe however that new design methodologies that start from high-level specifications and VHDL, as well as other new methodologies mentioned in the introduction, will create large initial machines with don't cares, so that the ideas presented in this paper will become even more useful for the design practice.

The conclusion of this paper is: "For many initial descriptions of sequential circuits that have a high percentage of don't cares, the two-dimensional FSM minimization/decomposition introduced here gives results superior to the FSM state minimization in the terms of total area of the main FSM and encoder realizations".

ACKNOWLEDGMENTS.

The authors would like to thank Professor Robert Brayton from University of California in Berkeley, Professor Maciej Ciesielski from University in Massachusetts in Amherst, and Professor Fabio Somenzi from the University of Colorado in Boulder for helping to find the respective literature, and especially the paper by Grasselli and Luccio [10]. Thanks are also to Prof. Somenzi for the access to his FSM minimizer and to Prof. Ciesielski for his set of synthesis tools. Special thanks are due to Dr. Tim Ross from Wright Labs, Professor Arkadij Zakrevskij from Belarussian Academy of Sciences, and Professor Shmerko from Technical University of Szczecin in Poland for providing us with examples of **practical** design methodologies that produce machines with very high percent of don't cares, many inputs, outputs, and compatible states.

References

[1] P. Ashar, S. Devadas and A.R. Newton, "A Unified Approach to the Decomposition and Re-decomposition of Sequential Machines," *Proceedings of the 27th A.C.M./I.E.E.E. Design Automation Conference,* pp. 601-606, 1990.

[2] P. Ashar, S. Devadas, and A.R. Newton, "Testability Driven Synthesis of Interacting Finite State Machines," *Proceedings of the International Conference on Computer Aided Design,* pp. 273-276, 1990.

[3] J. Barzdins, G. Barzdins, K. Apsitis, and U. Sarkans, "Towards efficient inductive synthesis of expressions from Input/Output examples," *Lecture Notes Notes in Comp.Sc.*, Vol. 744, Springer Verlag, pp.59-72, 1993.

[4] G. DeMicheli, R. Brayton, and A.L. Sangiovanni-Vincentelli, "Optimal State Assignment for Finite State Machines," *IEEE Transactions on Computer-Aided Design,* CAD-4, 269-285, 1985.

[5] G. DeMicheli, and T. Klein, "Algorithms for Synchronous Logic Design," *Proceedings of the International Symposium on Circuits and Systems,* pp. 756-761, 1989.

[6] S. Devadas, "Approaches to Multi-Level Sequential Logic Syntesis," *Proceedings of the 26th A.C.M./I.E.E.E. Design Automation Conference,* pp. 270-276, 1989.

[7] S. Devadas, and A. Newton, "Exact Algorithms for Output Encoding, State Assignment and Four-Level Boolean Minimization," *Proc. Hawaii Intern. Conf. on System Sciences,* pp. 387-396, 1990.

[8] A. Ghosh, S. Devadas, and A. Newton, "Verification of Interacting Sequential Circuits," *Proceedings of the 27th A.C.M./I.E.E.E. Design Automation Conference,* pp. 213-219, 1990.

[9] A. Grasselli, F. Luccio, "A Method for Minimizing the Number of Internal States in Incompletely Specified Sequential Networks," *IEEE Transactions on Electron. Computers,* Vol. EC-14, pp. 350-359, 1965.

[10] A. Grasselli, F. Luccio, "A Method for the Combined Row-Column Reduction of Flow Tables," *Proc. 7th Annual Symposium on Switching and Automata Theory,* New York, pp. 136-147, 1966.

[11] R.W. House and D.W. Stevens, "A New Rule for Reducing CC Tables," *IEEE Transactions on Computers,* Vol. C-19, pp. 1108-1111, 1970.

[12] B. Iyer, and M. Ciesielski, "Reencoding for Cycle-Time Minimization under Fixed Encoding," *Proc. ICCAD'98*, paper 6B.1, pp. 312 - 318.

[13] Z. Kohavi, *Switching and Finite Automata Theory,* New York, McGraw Hill, 1978.

[14] E.B. Lee, and M.A. Perkowski, "Concurrent Minimization and State Assignment of Finite State Machines," *Proc. of Intern. IEEE Conf. on Systems, Man, and Cybernetics,* Halifax, Nova Scotia, pp. 248-260, 1986.

[15] F. Luccio, "Reduction of the Number of Columns in Flow Table Minimization," *IEEE Transactions on Electronic Computers,* Vol. EC-15, pp. 803-805, 1966.

[16] F. Luccio, "Extending the Definition of Prime Compatibility Classes of States in Incomplete Sequential Machine Reduction," *IEEE Transactions on Computers,* Vol. C-18, pp. 537-540, 1969.

[17] D. Pager, "Conditions for the Existence of Minimal Closed Covers Composed of Maximal Compatibles," *IEEE Transactions on Computers,* Vol. C-20, pp. 450-452, 1971.

[18] M.C. Paul and S.H. Unger, "Minimizing the Number of States in Incompletely Specified Sequential Switching Functions," *IRE Transactions on Electron. Computers,* Vol. EC-8, pp. 356-367, 1959.

[19] M.A. Perkowski and N. Nguyen, "Minimization of Finite State Machines in System SuperPeg," *Proceeding of the Midwest Symposium on Circuits and Systems,* Luisville, Kentucky, pp. 139-147, 1985.

[20] M.A. Perkowski, J. Liu, and J.E. Brown, "Rapid Software Prototyping: CAD Design of Digital CAD Algorithms," *Progress in Computer-Aided VLSI Design Tools,* Vol. 1, G. W. Zobrist, ed., Ablex, pp. 353-401, 1989.

[21] M.A. Perkowski, and J. Liu, "Generation of Finite State Machines from Parallel Program Graphs in DIADES," *Proceedings of the I.E.E.E. International Symposium on Circuits and Systems,* pp. 1139-1142, 1990.

[22] M.A. Perkowski, and L.B. Nguyen, "An Encoding Program for Concurrent Finite State Machines Realized in Constrained Logic," *Proceed-*

ings of the Midwest Symp. on Circuits and Systems, Calgary, Alberta, Canada, pp. 204-207, 1990.

[23] M.A. Perkowski, P. Dysko, and B. Falkowski, "Two Learning Methods for a Tree-Search Combinational Optimizer," *Proc. Intern. Phoenix Conference on Computers and Communication*, Scottsdale, Arizona, March, pp. 606-613, 1990.

[24] M.A. Perkowski, M. Marek-Sadowska, L. Jóźwiak, T. Luba, S. Grygiel, M. Nowicka, R. Malvi, Z. Wang, and J.S. Zhang, "Decomposition of Multiple-Valued Relations," *Proc. ISMVL Conference*, pp. 13-18, 1997.

[25] M.A. Perkowski, L. Jóźwiak, and D. Foote, "Architecture of a Programmable FPGA Coprocessor for Constructive Induction Approach to Machine Learning and other Discrete Optimization Problems," *Proc. 4th Reconfigurable Architectures Workshop, 11th Intern. Parallel Processing Symposium,* Geneva, Switzerland, April 1-5, pp. 33-40, 1997.

[26] M.A. Perkowski, A.N. Chebotarev, and A.A. Mishchenko, "Evolvable Hardware or Learning Hardware? Induction of State Machines from Temporal Logic Constraints," *The First NASA/DOD Workshop on Evolvable Hardware (NASA/DOD-EH 99).* Jet Propulsion Laboratory, Pasadena, California, USA, July 19-21, 1999.

[27] I. Noda and M. Nagao, "A Learning Method for Recurrent Networks Based on Minimization of Finite Automata," *Proc. IJCNN'92*, Baltimore, pp. 127-132, Jun., 1992.

[28] J.K. Rho, G.D. Hachtel, F. Somenzi, and R. Jacoby, "Exact and Heuristic Algorithms for the Minimization of Incompletely Specified State Machines," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* Vol. 13, No.2, pp. 167-177, 1994.

[29] K.E. Stoffers, "Sequential Algorithm for the Determination of Maximum Compatibles," *IEEE Transactions on Computers,*, pp. 95-98, 1974.

[30] B. Trakhtenbrot and Ya. Barzdin', "Finite Automata: Behavior and Synthesis," *North-Holland Publishing Company,* Amsterdam, 1973.

[31] S. Yang, M. Ciesielski, "Optimum and Suboptimum Algorithms for Input Encoding and its Relationship to Logic Minimization," *IEEE Trans. on CAD,* Vol. 10, No. 1, Jan. 1991, pp. 4-12.

	Present Inputs					
PS	X_1	X_2	X_3	X_4	X_5	X_6
	000	10-	-11	001	110	010
1	1/- -	3/- -	1/0 -	3/- 0	3/- -	0/- -
2	1/- -	3/- 0	0/- -	1/- -	0/- -	4/- -
3	0/- -	0/- -	1/- 0	1/0 -	3/0 -	1/- -
4	4/- 0	3/- -	1/- -	3/- -	3/- 1	0/- -

NS/Present Outputs

Table 1

An Initial State Table of Mealy Machine M^0. Symbol 0 *denotes unspecified state transitions, "-" means an unspecified bit of data. The machine has three input bits and two output bits.*

a			b			c		
I^0	CFC^0	CGG^0	I^1	CCF^1	CGG^1	I^2	CFC^2	CGG^2
1	1	1	1	1	1	1	1	4
2	1	2	2	1	4	2	2	2,5
3	1	3	3	2	2,5	3	3	1,3,6
4	1	5	4	2	3,6	-	-	-
5	1	5	-	-	-	-	-	-
6	1	6	-	-	-	-	-	-

Table 2

The Process of the Input Groups Collection in List CGG. (a) at the beginning, (b) after the first column minimization (Table 4), (c) after the second column minimization (Table 6).

1,2	0	0	0	1	3	0	0	1	4
1,3	1	0	0	0	0	0	0	0	0
1,4	1	0	0	0	0	0	0	0	0
2,3	0	0	0	0	0	0	0	1	4
2,4	-1	1	4	1	3	0	0	1	4
3,4	0	0	0	1	3	0	0	0	0

Table 3

The Merger List COMPAT Created for Row Minimization from machine M^1 from Table 4.

	Present Inputs			
PS	X_1	X_4	$X_{2,5}$	$X_{3,6}$
	000	001	10-,110	-11,010
1	1/- -	3/- 0	3/- -	1/0 -
2	1/- -	1/- -	3/- 0	4/- -
3	0/- -	1/0 -	3/0 -	1/- 0
4	4/- 0	3/- -	3/- 1	1/- -

Table 4

The new state machine M^1 created from Table 1 after Column Minimization.

	Present Inputs			
PS	I_1	I_4	$I_{2,5}$	$I_{3,6}$
	000	001	10-,110	-11,010
1'	1'/- 0	2'/- 0	2'/- 1	1'/0 -
2'	1'/- -	1'/0 -	2'/0 0	1'/- 0

Table 5

The new state machine M^2 created from machine M^1 in Table 4 after Row Minimization.

	Present Inputs		
PS	X_4	$X_{2,5}$	$X_{1,3,6}$
	001	10-,110	000,-11,010
1'	2'/- 0	2'/- 1	1'/0 0
2'	1'/0 -	2'/0 0	1'/- 0

Table 6

The optimal state machine M after the second column minimization applied to machine M^2 from Table 4. This table cannot be further row minimized, which terminates the iteration of minimizations.*

inputs	m	n
X_4	0	0
$X_{2,5}$	0	1
$X_{1,3,6}$	1	0

Table 7

The Code Table Created from the Grouped Inputs. m and n are encoder outputs.

	sol	s03	mc	mab	za-mh4	mal-12	mal-14	shm-04
initial data format	.kis	.st	.kis	.kis	.st	.st	.kis	.kis
number of input bits	3	-	5	3	-	-	5	12
number of output bits	2	2	3	2	4	2	4	14
number of columns of M^0	6	4	24	6	20	8	32	49
number of rows of M^0	4	50	5	5	20	4	12	28
% of don't care NSs	25	21.2	40	43.33	74.7	62.5	43	80.4
% of don't care outputs	66.67	34.24	40	60	85	84.8	48	73.0
number of iterations	3	2	2	4	3	3	4	5
number of columns of M^*	3	4	19	3	11	3	17	21
number of rows of M^*	2	46	5	4	10	3	7	11
time of execution(sec)	1.3	7.4	8.0	1.5	10.3	1.2	9.2	24.2

Table 8

Minimization of benchmark machines with high percent of don't cares.

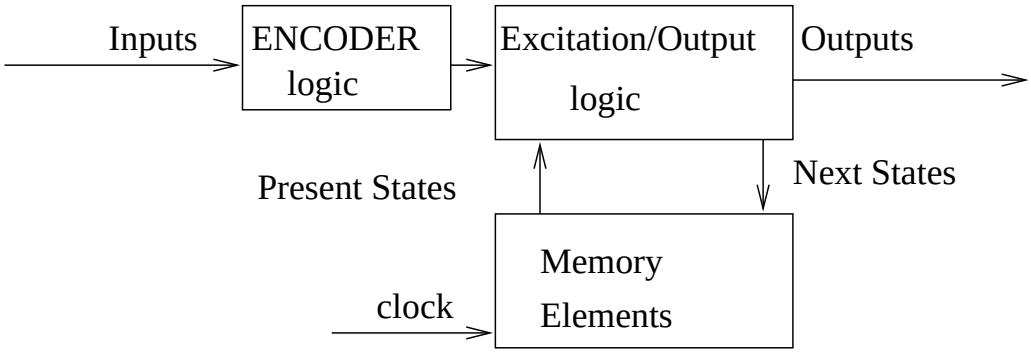

Fig. 1. *A partitioned realization of an FSM with an input encoder.*

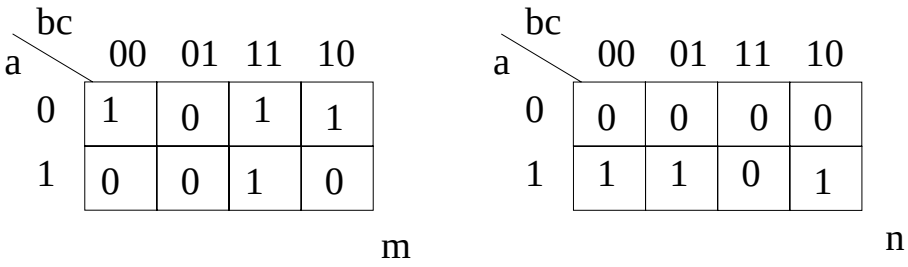

Fig. 2. *Explanation of the realization of the encoder for the decomposed realization of the FSM.*